Hierarchical Multithreading: Programming Model and System Software

Guang R. Gao[1], Thomas Sterling[2,3], Rick Stevens[4], Mark Hereld[4], Weirong Zhu[1]

[1]Department of Electrical and Computer Engineering
University of Delaware
{ggao,weirong}@capsl.udel.edu

[2]Center for Advanced Computing Research
California Institute of Technology
tron@cacr.caltech.edu

[3]Department of Computer Science
Louisiana State University
tron@cct.lsu.edu

[4]Mathematics and Computer Science Division
Argonne National Laboratory
{stevens,hereld}@mcs.anl.gov

Abstract

This paper addresses the underlying sources of performance degradation (e.g. latency, overhead, and starvation) and the difficulties of programmer productivity (e.g. explicit locality management and scheduling, performance tuning, fragmented memory, and synchronous global barriers) to dramatically enhance the broad effectiveness of parallel processing for high end computing. We are developing a hierarchical threaded virtual machine *(HTVM) that defines a dynamic, multithreaded execution model and programming model, providing an architecture abstraction for HEC system software and tools development. We are working on a prototype language, LITL-X (pronounced "little-X") for Latency Intrinsic-Tolerant Language, which provides the application programmers with a powerful set of semantic constructs to organize parallel computations in a way that hides/manages latency and limits the effects of overhead. This is quite different from locality management, although the intent of both strategies is to minimize the effect of latency on the efficiency of computation. We will work on a dynamic compilation and runtime model to achieve efficient LITL-X program execution. Several adaptive optimizations will be studied. A methodology of incorporating domain-specific knowledge in program optimization will be studied. Finally, we plan to implement our method in an experimental testbed for a HEC architecture and perform a qualitative and quantitative evaluation on selected applications.*

1 Introduction

With the rapid increase in both the scale and complexity of scientific and engineering problems, the computational demands grow accordingly. Breakthrough-quality scientific discoveries and optimal engineering designs often rely on large scale simulations on High-End Computing (HEC) systems with performance requirement reaching peta-flops and beyond. However, current HEC systems lack system software and tools optimized for advanced scientific and engineering work of interest, and are extremely difficult to program and to port applications to. Consequently, applications rarely achieve an acceptable fraction of the peak capability of the system.

To radically improve this situation, the following key features are expected to be supported in the future HEC systems: (1) Architecture support for coarse- and/or fine-grain multithreading at enormous scale (up to millions of threads). (2) Architecture support for runtime thread migration, and (3) Architecture support for large shared address space across nodes. These features can be observed in the IBM Bluegene L [6] and Cyclops architectures [5], Processor-In-Memory-based architectures [17], fine-grain multithreaded architectures like HTMT [10] and CARE [14].

In this paper, we propose a *hierarchical threaded virtual machine* (HTVM) that defines a dynamic, multithreaded execution model, which provides an architecture abstraction for HEC system software and tools development. A corresponding programming model will efficiently exploit the ability of the execution model by users. We will perform research on programming model

1-4244-0054-6/06/$20.00

and language issues, continuous compilation and runtime software that are critical to enable the dynamic adaptation of the HEC system. We propose a method to enable domain-expert knowledge input and exploitation, and runtime performance monitoring mechanism to support the above continuous compilation. Finally, we report the current status of the implementation, performance analysis, and evaluation of the proposed methods under an experimental HEC system software testbed.

2 An Overview of the Hierarchical Threaded Virtual Machine Model

This section gives our overall vision on the HEC system software/tools. A major challenge is to accommodate dynamic adaptivity in the design due to the complex and dynamic nature of ultra-large scale HEC applications and machines. Under a real HEC program execution scenario, millions of threads at various levels in the thread hierarchy may be generated and executed at different time and places in the machine. Each thread should be mapped to a desirable physical thread unit when resources become available and dependences are resolved. We identify four classes of adaptivity critical to the performance of the system:

- **Loop parallelism adaptation**. Scientific applications tend to have computation-intensive kernels consisting of loop nests. Exploitable parallelism in a loop nest, and the grain size of the parallelism, are runtime dependent on the machine resource availability and data locality, which change more drastically in a highly threaded environment with deep memory hierarchy.
- **Dynamic load adaptation**. The computation load may become unbalanced and a large number of threads may need to migrate to balance the load of the machine.
- **Locality adaptation**. Data objects may need to migrate, and copies be generated and moved in the memory hierarchy to achieve high locality, while copy consistency needs to be preserved.
- **Latency adaptation**. The deep memory hierarchy usually found in an HEC machine makes the memory access latencies vary more drastically during the execution, depending on the locality of references, the number of concurrent accesses, and the available memory bandwidth. The system needs dynamically adapt to such variations.

A main task of our research is to study the key system software technologies that support the above dynamic adaptiveness of the HEC system. Fig. 1 shows our overall system software architecture. At the core is a Hierarchical Threaded Virtual Machine(HTVM) execution model that features dynamic multi-level multithreaded execution. HTVM includes three components: a thread model, a memory model and a synchronization model. This design focuses on adaptivity features, as will be discussed in detail in Section 3. The functionalities of HTVM will be supported and explored through the HTVM parallel programming language (called LITL-X), compiler and runtime software. The compiler has two parts: a static part and a dynamic part. As shown in Fig. 1, the dynamic compiler is responsible for the adaptation of loop parallelism, dynamic load, locality and latency. Since the dynamic compiler closely interacts with the runtime system and will be called during the execution of the HEC applications, its functionality extends smoothly to the runtime system as well, as indicated by the boxes that span across the dynamic compiler and the runtime system.

To take advantage of the adaptivity features of HTVM more effectively, a *domain-experts knowledge base* is provided. Domain-specific knowledge is expressed as scripts, which give specific annotations to the source of the HEC applications to guide the compilation process of the static compiler. To assist adaptivity, a system of *structured hints* guides the dynamic compiler for selection and completion of the *partial schedules* generated by the static compiler, and for selection of *runtime algorithms*, based on the *dynamic facts* such as memory access patterns found by a runtime performance monitor during the execution of the HEC applications. The flow of the mapping process of an HEC application under our proposed research software and tools is indicated by the big shaded arrows in Fig. 1. The components of the software infrastructure have been annotated by the corresponding section numbers. The HTVM compilation and execution process is an iterative process with the assistance of a feedback process as shown in the figure.

3 Hierarchical Multithreading: Programming Model and System Software

3.1 A Hierarchical Threaded Virtual Machine Model

One of our primary objectives is to define the hierarchical threaded virtual machine (HTVM). We first outline our research in the HTVM execution model, which consists of a thread model, a memory model and

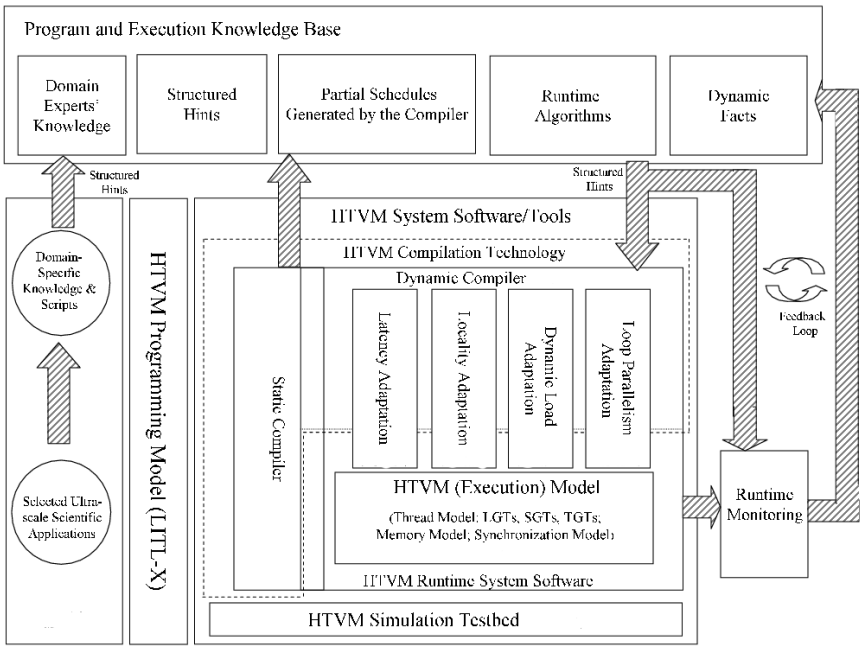

Figure 1. An Overview of the Proposed HEC Software/Tools

a synchronization model. We then outline research issues and tasks for HTVM programming model.

3.1.1 HTVM Execution Model

A novel aspect of our HTVM model is to provide a smooth and integrated abstraction that directly represents these thread levels and provides an integrated thread hierarchy. We will target future HEC architectures - they provide rich hardware support for a hierarchy of threads at different grain levels, as discussed earlier.

Intuitively, the following levels of threads are to be defined under HTVM.

- **Large-Grain Threads (LGTs) under HTVM**. Large-grain threads are a universally supported feature of many HEC architectures. These threads normally perform a substantial computation task, building up their state, of considerable "weight", during the course of their execution. There is usually considerable cost associated with such a coarse thread invocation and management, even with architectural support. Examples of LGTs are the high-weight threads under Cascade architecture [4] or coarse-grain threads under PERCS architecture [1], and the threads under Cyclops-64 TiNy thread^{TM} [7].
- **Small-Grain Threads (SGTs) under HTVM**: Small-grain threads are another feature of certain HEC architectures interested in this proposal. These threads normally expect to perform a much smaller computation task, building some state but with substantial less "weight". Therefore, cost of their invocation and management is much lower when comparing with large-grain threads. An example of SGTs is the threaded function calls under CILK [9] and EARTH[19], *parcels* under HTMT [10] and Cascade [4], and asynchronous calls being considered under PERCS [1].
- **Tiny-Grain Threads (TGTs) under HTVM**: threads with much lighter weight than SGTs will be supported in some future HEC architectures. The partition of TGTs and their resource usage (e.g., registers) are done by automatic thread partitioning [18]. Examples of TGTs include *fibers* under EARTH [19] and *strands* under CARE [14].

An important research task is to provide a solid definition and specification of the three levels of threads under a unified thread hierarchy. The specification needs to be general enough to capture the features of a family of future HEC architectures to ensure portability, while simple enough for compiler and/or programmers to generate efficient code, and to facilitate runtime optimization.

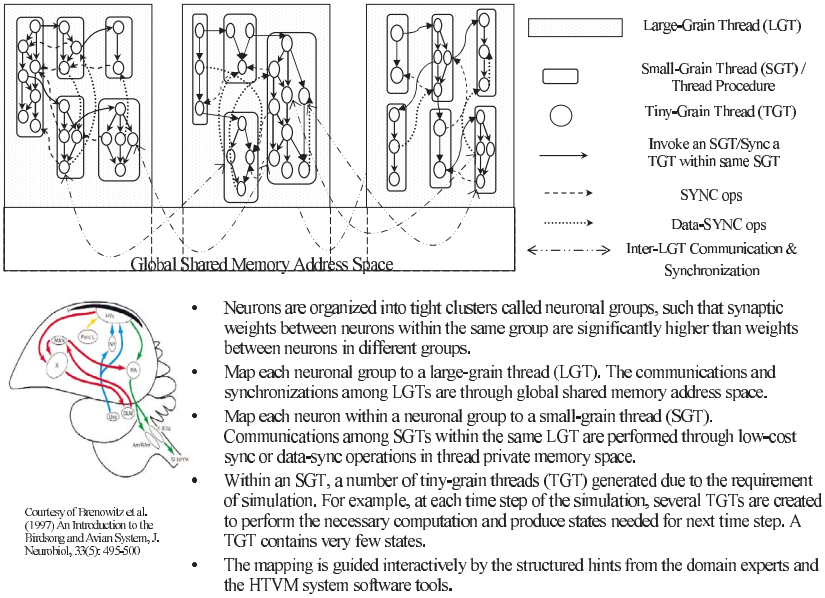

Figure 2. A Case Study of Hierarchical Thread Execution Model: Large Scale Simulation of Brain Neuron Networks

Our current plan is: An LGT has its own private memory space, and all LGTs share a global address space. A group of SGTs invoked from an LGT will see the private memory of the LGT. An SGT invocation will have its own private frame storage, where its local state is stored. The TGTs within an SGT will share the frame storage of the enclosing SGT invocation, but may communicate efficiently by using registers under the compiler control. To illustrate our thoughts, Figure 2 shows a mapping of the computation of a multi-level neural system simulation onto our HTVM hierarchy of threads. We hope the figure should be self-explanatory.

3.2 Parallel Programming Model and LITL-X

For developing the parallel programming model for HTVM, we leverage our own experience in participating recent on-going research on new parallel programming models and languages such as the language proposal X10 under the IBM PERCS project [2] and Chapel under the Cray Cascade project [4]. All are seeking a potential alternative programming model that is more aggressive in addressing the combined challenges of latency and overhead.

To be more concrete, We are working on a prototype language, LITL-X (pronounced "little-X") for Latency Intrinsic-Tolerant Language, which provides the application programmers with a powerful set of semantic constructs to organize parallel computations in a way that hides/manages latency and limits the effects of overhead. This is quite different from locality management, although the intent of both strategies is to minimize the effect of latency on the efficiency of computation. Locality management attempts to *avoid* latency events by aggregating data for local computation and reducing large message communications. Latency management attempts to *hide* latency by overlapping communications with computation.

LITL-X will incorporate the following classes of parallel constructs for latency tolerance and overhead reduction:

- *Coarse-grain multithreading*, with thread context-switching built in the application's instruction stream (rather than in the operating system) for keeping the processors busy in the presence of remote requests. This is connected to the LGT under HTVM.
- *Parcel(intelligent messages)-driven split-transaction computation* [17], to reduce communication and to enable the moving of the work to the data (when it makes sense). This is connected to the SGT under HTVM.

- *Futures* [11] for eager producer-consumer computing, with efficient localized buffering of requests at the site of the needed values. This is connected to the TGT under HTVM.
- *Percolation* [12] of program instruction blocks and data at the site of the intended computation, to eliminate waiting for remote accesses, which are determined at run time prior to actual block execution.
- *Synchronization constructs* for data-flow style operations, as well as *atomic blocks of memory operations*.

3.3 System Software: Compiler and Runtime Solutions

In this section, we describe how the compiler and runtime software to address the challenges of efficient execution under the HTVM model. Our solution is moving from static analysis and optimization toward a hybrid scheme, combining both static compilation and runtime adaptation. The compiler and the runtime system software are intimately connected under our adaptive/continuous compilation strategy, where some key functions of runtime system software can also be viewed as an extension of the compiler. As mentioned in Section 2, the HTVM system software addresses four types of runtime adaptation: loop parallelism adaptation, dynamic load adaptation, locality adaptation, and latency adaption. In this paper, we take the loop multithreading and parallelism adaptation as an example to illustrate how the system software should be designed.

Scientific applications heavily rely on loop nests to compute their results. Often, more than 90% of the execution time is spent on some computation-intensive kernels composed of loop nests. It is of extreme importance to schedule these loops effectively to improve the overall performance of the application. Loop scheduling on a parallel distributed system can be broadly divided into two classes: static and dynamic scheduling. Static scheduling tends to cause load imbalance, since the exploitable parallelism, and the grain size of the parallelism, vary with the machine resource availability, data distribution and the latency of memory accesses, especially in the context of the highly dynamic and threaded HEC machines. Consequently, dynamic scheduling has been developed and shown promising performance improvement.

The dynamic loop scheduling methods, however, target only *Thread-Level Parallelism (TLP)*. In contrast, there is another important technology, namely, *software pipelining*, aims to exploit *Instruction-Level Parallelism (ILP)* from loops. Software pipelining is a most widely and successfully used loop parallelization technique for existing microprocessor architectures (e.g. VLIW or superscalar architectures) [13]. Traditionally, software pipelining is mainly applied to the innermost loop of a given loop nest. Recently we have introduced a new approach, called Single-dimension Software Pipelining (SSP) [16],to software pipeline a loop nest at an *arbitrary* loop level with desirable optimization objectives such as data locality and/or parallelism. The SSP method has been successfully tested on a uniprocessor architecture (Intel IA-64 architecture) and shows significant performance improvement.

In this research, we will further extend SSP from single-processor single-thread environments to multiprocessor multithreading environments, by combining the strength of software pipelining (a static scheduling) and dynamic scheduling. The basic approach can be described as follows: First choose the most profitable loop level [16], which may have its own inner loops and therefore a loop nest itself. This loop level is software pipelined first. After that, the software pipelined code is partitioned into threads, each thread composed of several iterations of the selected loop level. The approach is unique in that it exploits instruction-level and thread-level parallelism simultaneously.

There are several issues we need to study: (1) What is the performance and cost model for such partition of the software pipelined code into threads? (2) How to integrate this approach with runtime optimization? Software pipelining uses a machine resource model, including the memory access latencies, to scheduling the loop. The available resources and actual memory access latencies, however, are runtime dependent in an HEC machine, as explained before. (3)What semantics constructs can be provided in LITL-X specifically for SSP and multi-threading? For example, a pragma may be presented to indicate the most beneficial loop level, or indicate the scheduling strategies. The static compiler acts according to the pragma and generates some (partial) schedules, and stores this pragma as a structured hint in appropriate format if it is dependent on runtime statistics.

4 Efficient Interaction between Applications and System Software

In this section, we outline the solution strategies for the efficient interaction between applications and system software. There are two important aspects: the first is developing the methods, models, and tools to facilitate mapping complex domain application codes to the HTVM model; the second is developing a mon-

itoring methodology and interface that will help the adaptive compiler and runtime system to optimize execution and resource utilization on the fly.

4.1 Domain-Specific Knowledge Input to System Software

Today, it is a widely accepted belief that some ultra-large scale scientific applications targeted by the HEC machines are so complex that efficient mapping of such applications to the architecture will require application scientists/programmers who are domain experts. The gap between expressions of domain-specific computations and expressions tailored to efficient execution on a given system architecture is widening. Domain experts are and will continue to be challenged to write high performance codes.

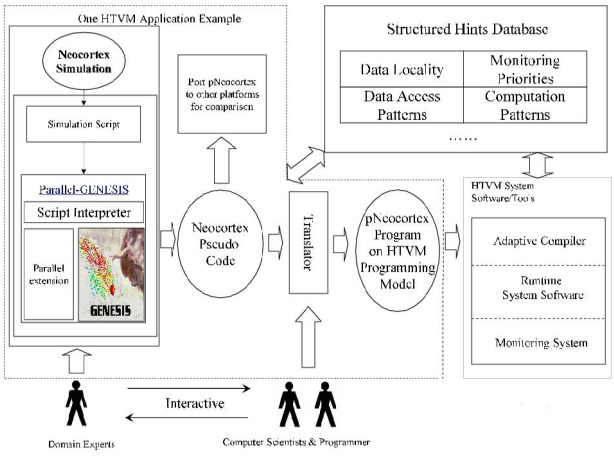

Figure 3. Mapping pNeocortex to system software

HTVM embodies a model that we believe expresses effectively the low-level idioms and interfaces required by future ultra-large scale computing platforms. But programming at the HTVM level will require expertise well outside the domain of typical application specialists. To bridge this gap with as little compromise as possible we present a layered architecture that we will implement manually first, with an eye toward later automation as relevant technologies mature. Fig.3 illustrates the basic idea as it has evolved in the context of the EARTH project [19] and a particular application: a simulation of electrical activity in the neocortex. With the assistance from the domain specific knowledge embedded in the neocortex simulation, a test model of the PGENESIS neocortex is designed and the code mapping and optimization on the EARTH base programming model and runtime system software are under the guidance of the domain experts' knowledge. It shows the progression from a domain-specific script-based description of a simulation to HTVM code. The domain expert's knowledge is built into the script language and/or idiomatic modules that augment the script or other programming language. Pseudo-code distills the simulation down to its key structural and computational components, and includes hints to be used to guide optimization. This code is then translated to run on the HTVM. The resulting code is ready for compilation and execution.

Guidance from the application programmer, and more generally available from domain-specific idioms and algorithms used explicitly or implicitly by the application programmer, must be passed to the adaptive compiler, runtime system, and monitoring system to enable them to efficiently optimize the execution of the code. We plan to define and implement a system of *structured hints* to capture and apply the combined expertise of the domain specialist and the compiler. Our notion of *structured hints* embodies the idea that the compiler and the domain expert can collaborate to reduce the number of possible optimization strategies to a modest and manageable set of options which are most likely to produce high performance code in the context of a complex and adaptive system architecture.

- The compiler will identify points in the code which present the potential for optimization, but for which it has insufficient information to proceed on its own.
- The domain expert, led by the structured list of opportunities generated by the compiler, will add priorities and rules to this list of opportunities that will aid the compiler and runtime to streamline code execution.

The resulting organized and expertly culled guide to optimization, the structured hints, includes data structures, dependencies, weights, and rules. In addition to focusing the compilers attempts at optimization, the resulting structured hints will be an integrated part of our Program/Execution Knowledge Database, providing the runtime system with an informed and tailored set of options around which to make its choices. Each hint can be expressly targeted at some part of the execution model: the adaptive compiler, the runtime system, or monitoring system. For example, informed choices about which pieces of the code to instrument, and how, will become part of the metric suite used by the adaptive compiling and runtime system to adjust resource allocation and compilation strategy during execution. As another example, the domain expert can

identify critical parameters to be adjusted by the compiler for its adaptive optimizations, thereby narrowing the parameter space to be searched. Without reference to the underlying hardware architecture, or even to the HTVM software architecture, the hints must address, in a general way, issues of: 1) data locality, 2) monitoring priorities, 3) data access patterns, and 4) computation patterns. These will be mapped directly to specific actions, weighting schemes, and optimization strategies in the HTVM system software.

4.2 Monitoring of Application Execution

The adaptive compile and runtime system will require feedback derived from the execution and resource allocation monitoring. The hints discussed in the previous subsection will drive both static and dynamic optimizations of the program execution. In this later context, they will provide the system with guidance on degrees of freedom most likely to affect performance, likely bottlenecks in the code, unpredictable aspects of data locality and computational work patterns to steer monitoring resources to develop heuristic models.

Our plan is to implement a hint schema that is fed by the application programming and domain-expert interactions. It will be used by various stages of the code translation process, the HTVM system, and the runtime monitoring system.

5 Infrastructure and Experimentation Plan and Status

In this section, we report the current status of the implementation and experimentation of the system software and tools.

5.1 The Infrastructure of an Experimental Testbed

For the implementation and experimental study of the HTVM system software, we continue to develop and refine our software infrastructures. As the starting point, we choose the system software infrastructure for IBM Cyclops-64 cellular architecture, a petaflops supercomputing chip-multithreaded architecture under development at IBM Research Laboratory, featured with 160 thread units and a number of memory modules interconnected by a high speed on-chip interconnection network [8]. We will leverage an system software infrastructure and tool-chain for this architecture being developed jointly by a collaborative effort between ETI and CAPSL at University of Delware. The system software infrastructure includes a runtime system for a threaded virtual machine at LGT level, a thread communication and synchronization library, a OpenMP compiler and runtime system, a function-accurate simulator, a cycle-accurate simulator, a Gcc-based compiler, the binary utilities (assembler, linker, etc.), and the libraries (e.g. libc/libm). We also continue develop the EARTH and CARE software infrastructure at SGT and TGT levels. We have re-targeted the Open64 compilation infrastructure from the 64-bit Intel IA-64 architecture to the 32-bit Intel XScale embedded architecture [3], and recently we have successfully implemented the SSP scheduling [16], register allocation, and code generation [15] in this compiler.

Based on the above infrastructures, we are constructing an experimental testbed in the following way. First we are modifying the current virtual machine that is for large-grain threads under C64 software infrastructure, and implement the HTVM small-grain and tiny-grain threads. Second we are implementing runtime system software support for both, leveraging our experience with the EARTH and CARE software infrastructures. Third, we are extending and modifying the function-accurate simulator to include the support of relevant architecture features. We are also extending the above runtime system, compiler and simulator to implement the algorithms developed during this research for continuous compilation and runtime optimization, as introduced in Section 3.3.

5.2 Experimentation Plan and Status

The primary goal of the experimentation is to validate the proposed HEC system software and tools in addressing the needs of selected HEC applications. Have we created a practical methodology, ultimately amenable to automation, that enables efficient application programming by domain experts while producing codes that perform well? We have selected two codes for our study: the *computational neuroscience*, which simulates large networks of biological neurons, and the *fine grain molecular dynamics*, which simulates relatively modest sized molecules, a single protein or protein complex in water with multiple ion species. Both codes are representative HEC applications, therefore they are important for the research and development of HTVM execution model, programming model, and system software.

Our proposed experimental methodology comprises five major tasks enumerated below. We will use our Neuroscience code to blaze the trail and follow the task list for the molecular dynamics code with a delay. In this way we will begin the development and testing of the process and tool set with one code, and use the

second code to validate the process and provide opportunity for refinement.

- Instrument and characterize the application codes on existing machines to establish base performance properties.
- Develop performance models for each code in terms of the proposed HTVM model.
- Develop a new implementation of each code to use the proposed HTVM model using the application mapping methodology.
- Validate on the simulation testbed.
- Project performance and impact of the proposed new HEC software and tools.

6 Acknowledgments

We acknowledge the support from the National Science Foundation (CNS-0509332) and Laboratory Director Research and Development funding, IBM, ETI, and other government sponsors. We acknowledge Hongbo Rong, who has been instrumental in the development and documentation of some key ideas in this paper. We would also like to acknowledge other members at the CAPSL group, who provide a stimulus environment for scientific discussions and collaborations, in particular Ziang Hu, Juan del Cuvillo, and Ge Gan.

References

[1] DARPA: High Productivity Computing Systems (HPCS).

[2] IBM: PERCS (Productive, Easy-to-use, Reliable Computing System).

[3] Kylin C Compiler.

[4] D. Callahan, B. L. Chamberlain, and H. P. Zima. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Santa Fe, New Mexico, April 26th 2004.

[5] C. Cascaval, J. Castanos, L. Ceze, M. Denneau, M. Gupta, J. M. D. Lieber, K. Strauss, and J. H.S. Warren. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA)*, Boston, Massachusetts.

[6] K. Davis, A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, S. Pakin, and F. Petrini. A performance and scalability analysis of the BlueGene/L architecture. In *Proceedings of SC2004: High Performance Networking and Computing*, Pittsburgh, PA, Nov. 2004. ACM SIGARCH and IEEE Computer Society.

[7] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. TiNy Threads: A thread virtual machine for the Cyclops64 cellular architecture. In *Fifth Workshop on Massively Parallel Processing, in conjuction with 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, page 265, Denver, Colorado, USA, April 2005.

[8] J. B. del Cuvillo, Z. Hu, W. Zhu, F. Chen, and G. R. Gao. Toward a software infrastructure for the cyclops64 cellular architecture. *CAPSL Technical Memo 55*, April 26th 2004.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[10] G. Gao, K. Theobald, A. Marquez, and T. Sterling. The htmt program execution model. *CAPSL Technical Memo 09*, July 1997.

[11] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[12] A. Jacquet, V. Janot, R. Govindarajan, C. Leung, G. Gao, and T. Sterling. Executable performance model and evaluation of high performance architectures with percolation. Technical Report 43, Newark, DE, Nov. 2002.

[13] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328, Atlanta, Georgia, June 22–24, 1988. *SIGPLAN Notices,* 23(7), July 1988.

[14] A. Marquez and G. R. Gao. CARE: Overview of an adaptive multithreaded architecture. In *Fifth International Symposium on High Performance Computing (ISHPC-V)*, Tokyo, Japan, October 20-22, 2003.

[15] H. Rong, A. Douillet, R. Govindarajan, and G. R. Gao. Code generation for single-dimension software pipelining of multi-dimensional loops. In *Proc. of the 2004 Intl. Symp. on Code Generation and Optimization (CGO)*, pages 175–186, Palo Alto, California, March 2004.

[16] H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software-pipelining for multi-dimensional loops. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 163–174, Palo Alto, California, March 2004. Best Paper Award.

[17] T. Sterling. An introduction to the gilgamesh PIM architecture. *Lecture Notes in Computer Science*, 2150, 2001. LNCSD9.

[18] X. Tang and G. R. Gao. Automatically partitioning threads based on remote paths. Technical Report 23, Newark, DE, July 1998.

[19] K. B. Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, May 1999.